NAVAL POSTGRADUATE SCHOOL
Monterey, California

19981023 029

THESIS

AN IMPLEMENTATION OF SECURE FLOW TYPE INFERENCE FOR A SUBSET OF JAVA

by

Ismail Okan Akdemir

September 1998

Thesis Advisor: Dennis Volpano

DTIC QUALITY INSPECTED 4

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE September 1998	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE AN IMPLEMENTATION OF SECURE FLOW TYPE INFERENCE FOR A SUBSET OF JAVA	5. FUNDING NUMBERS
6. AUTHORS Akdemir, Ismail Okan	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT(*maximum 200 words*)

Smart cards play an important role in a digital society. A smart card contains memory or an embedded microprocessor with the capability of enabling a wide variety of services, such as electronic cash in the case of memory cards and digital signature computation in the case of processor cards. A processor card can require a cardholder to authenticate herself in order to prevent others from using the card's services, from forging the cardholder's signature, for example. Authentication can be done by storing a personal identification number (PIN) or digitized fingerprint of the cardholder on the card itself. The PIN or fingerprint must always remain confidential no matter how the card is (ab)used.

This thesis addresses the problem of preserving the privacy of information stored on smart cards. Volpano and Smith have developed a static analysis for analyzing source code for information flow violations. This technique is developed further here for a language called Java Card, in which smart card applications are written. A prototype analyzer is presented for a subset of Java Card and applied to a sample card application to demonstrate its utility in protecting private information stored on smart cards.

14. SUBJECT TERMS Java Card, Smart Cards, Secure Flow Analysis, Type System			15. NUMBER OF PAGES **50**
			16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

AN IMPLEMENTATION OF SECURE FLOW TYPE INFERENCE FOR A SUBSET OF JAVA

Ismail Okan Akdemir
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy, 1992

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE
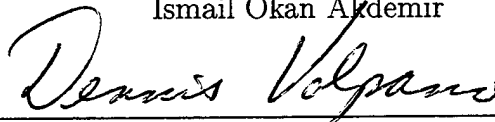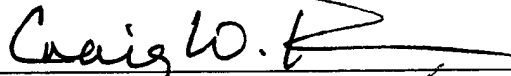
from the

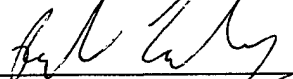**NAVAL POSTGRADUATE SCHOOL**
September 1998

Author: _____
Ismail Okan Akdemir

Approved by: _____
Dennis Volpano, Thesis Advisor

_____
Craig Rasmussen, Second Reader

_____
Dan Boger, Chairman
Department of Computer Science

iii

ABSTRACT

Smart cards play an important role in a digital society. A smart card contains memory or an embedded microprocessor with the capability of enabling a wide variety of services, such as electronic cash in the case of memory cards and digital signature computation in the case of processor cards. A processor card can require a cardholder to authenticate herself in order to prevent others from using the card's services, from forging the cardholder's signature, for example. Authentication can be done by storing a personal identification number (PIN) or digitized fingerprint of the cardholder on the card itself. The PIN or fingerprint must always remain confidential no matter how the card is (ab)used.

This thesis addresses the problem of preserving the privacy of information stored on smart cards. Volpano and Smith have developed a static analysis for analyzing source code for information flow violations. This technique is developed further here for a language called Java Card, in which smart card applications are written. A prototype analyzer is presented for a subset of Java Card and applied to a sample card application to demonstrate its utility in protecting private information stored on smart cards.

TABLE OF CONTENTS

LIST OF FIGURES

x

ACKNOWLEDGMENTS

I would like to thank Dr. Dennis Volpano for his help and guidance during this thesis. His deep knowledge helped me to learn how to question and explore a subject.

I would also like to thank Dr. Craig Rasmussen for his assistance and contributions to this thesis.

I also want to express my appreciation to Rena Henderson for her prompt responses and careful editing.

I.   INTRODUCTION

Magnetic-stripe cards with an embedded microprocessor have been around for many years. They are typically called smart cards and come in two varieties: memory cards and processor cards. A processor card has roughly the computing equivalent of a vintage 1980 personal computer with an 8-bit microprocessor, 512 bytes of RAM and 16K bytes of ROM. It can run applications and make decisions, unlike plain magnetic-stripe cards.

The first smart cards in wide-spread use were simple "memory", or stored-value, cards that served as electronic "purses" for small purchases. French banks were the first to introduce these cards, and the European telephone companies used them in place of coins in public telephones. Some stored-value cards cannot be recharged and are discarded after they are depleted. Others can be replenished with cash value at ATM-like stations. Fast-food giant McDonalds (tm) of Germany, for instance, allows customers to purchase food with rechargeable, stored-value cards. And if your card needs to be recharged, McDonalds provides automated tellers within its restaurants for this purpose. Smart cards have also been adopted for use in Europe's Global System of Mobile Communications (GSM). GSM cell-phone manufacturers have decided to equip handsets with smart-card readers. Users store their identity on smart cards and are able to use any such equipped cell phone. This is in contrast to having phones with their own identities, say a transmission frequency fingerprint, which, if intercepted, can lead to cell-phone cloning.

Smart cards obviously must store private information to be useful. For instance, a smart card user typically authenticates herself using a personal identification number (PIN) entered by a keypad attached to a card reader. The number is compared with a PIN stored on the card by code that is part of the card's operating system. This code and any other application running on the card that can inspect the stored PIN may (covertly) leak the PIN without knowing it. This occurs through

what are called covert channels, and there are many different kinds of such channels. Volpano and Smith address limiting the bandwidth of these channels through statically analyzing the source code of smart-card applications. This thesis explores extending their work to Java Card, a particular programming language embraced by some smart-card vendors for writing portable smart-card applications.

A. THE ARCHITECTURE OF A SMART CARD

Smart cards come in two basic flavors, memory and processor cards. They are the size of conventional magnetic stripe cards, but each carries a chip, which makes it smarter and more valuable. A card's architecture is given in terms of its hardware and software. On the software side, there is the runtime environment afforded applications running on the card. Hardware characteristics across cards tend to vary far less than do the runtime environments. Below are some examples of different hardware configurations.

1. Integrated Circuit(IC) Microprocessor Card: a card with 8-bit processor, 16 KB read-only memory (ROM),and 512 bytes of random access memory (RAM)

2. Integrated Circuit(IC) Memory Card: a card with 1-4 KB of data storage

3. Optical Memory Card: a card combined with a Compact Disk that can hold 4 MB of data

Smart cards also vary according to their operating, or runtime, environments. A runtime environment provides an interface between the card and the card terminal known as the card acceptance device (CAD). A CAD inputs requests from a card holder and is the card's interface to the outside world. Runtime environments typically vary across smart card vendors. Examples include the following.

1. PS/SC is one of the proprietary runtime environments supported by Microsoft. This environment currently supports only the Win32-based platforms.

2. OpenCard Framework is an other specification for the runtime environment. It is an open standard designed to provide interoperability of smart card applications across network computers, desktops, laptops, set tops, and so on.

3. JavaCard is another runtime environment. It was introduced by Schlumberger and submitted as a standard by JavaSoft. JavaCard is a set of standard classes, which is a subset of standard Java. JavaCard is designed to develop secure and hardware-independent applications. These applications consist of Applets that are quite similar to the Applets that run within a web browser.

Smart cards have the potential for a wide range of applications. For example, today we use different cards for different purposes, such as shopping and on-line transactions. A smart card can integrate these different areas and serve as a bridge between consumer electronic devices and the card holder. It might seem that such a tiny card would not have much functionality, but the point is not what the card does, but rather the kinds of services it enables. The idea is not to run big applications, but to provide access to them. We call these applications services. Other examples of the services enabled by smart cards include:

- Cable TV authentication

- Storage Internet addresses as bookmarks

- USPS certified e-mail

- Subscriptions to several advanced news services

- Subscriptions to some pay-per-view video streaming channels

- Ticketing

A smart card can provide a single interface to many services; however, there are security risks. Quite often a smart card must handle personal information in order to enable the services listed above. Depending on the service, some of this information may be stored on the card. It is important that the privacy and integrity of such information be preserved. As a simple example, suppose a program running on a smart card has the property that it throws a particular kind of exception if and only if some private bit is on (By the way, this property may be unkown to the programmer.). If the exception is not caught and handled, it will reach the card's interface and will be observable by any card user. So its presence, or absence, reveals the private bit.

B.   SCOPE OF THESIS

This thesis explores an approach to enforcing privacy in software using a type system, an idea originating with Volpano and Smith [Ref. 1]. It is concerned with adapting an earlier type system of theirs to Java Card 2.0, an object-oriented language for smart card applications. Java Card is interesting because applications (Applets) can run on any card with the Java Card Runtime Environment (JCRE) installed. This is a much more open framework. The new type system requires changes to their original type inference algorithm. This thesis addresses these changes. Obviously, there are other security issues for smart cards, such as the integrity of information stored on the card. This thesis focuses on privacy only. In an earlier thesis, Harvey [Ref. 2] adapted the type system of Volpano and Smith to address some features of Java Card 2.0. However, his thesis does not treat objects nor consider function methods.

We begin by exploring Java Card 2.0 and the Java Card Virtual Machine specifications. Then, in Chapter III, we give the core language for which the type system is developed. It is close to full Java Card 2.0. The type system for enforcing privacy is also given. In Chapter IV, we give a type inference algorithm for deciding whether programs have types in the system of Chapter III. It is called the secure flow analyzer. An application of the secure flow analyzer is given on a small Applet in Chapter V.

II. JAVA CARD AND THE VIRTUAL MACHINE

The widespread use of smart cards has been hampered by the traditional practice of smart card vendors. Each would embed its card's operations in its own proprietary operating system (OS), making it difficult to extend a card's functionality or to port that functionality to another card. There was a need to separate the card's OS from its applications and to develop those applications in a way that would allow them to be portable across a wide variety of smart cards. Recognizing this need, Visa International teamed up with Integrity Arts to design an open platform for smart cards in 1995. Simultaneously, the French smart-card maker Schlumberger was developing an architecture for the same purpose. The Schlumberger architecture resembled Sun's Java platform, which was being released as JDK 1.0 at the time. There was enough of a similarity that Visa and Schlumberger convinced Sun Microsystems to develop a specification of the smart card open platform. Java Card was born. The Java Card 1.0 specification was released in October 1996, and version 2.0 was released a year later. Version 2.1 is expected to be publicly available by October 1998.

Though based on Java, Java Card 2.0 differs from Java in many ways. The Java Card 2.0 language is used basically to program smart cards that implement the Java Card Runtime Environment (JCRE). The language and runtime environment are based on the Java programming language and the Java Virtual Machine (JVM). Java Card and the JCRE are described in [Ref. 3]. The most important ways that Java Card differs from Java are highlighted below:

1. Dynamic class loading: A Java Card System is not capable of loading classes dynamically. The transfer of the class files is done statically either during or after production of the card.

2. Security Manager: The security model of Java Card is considerably different from standard Java. There is no customizable security manager. Security policies are encoded in the virtual machine.

3. Threads: There is only one thread of control in a Java Card System. Neither Thread class in Java nor any structure about threads can be used in Java Card.

4. Cloning: Objects in a Java Card system are not cloneable. The base Object class does not implement the clone() method, and there is also no Cloneable interface.

5. Garbage Collection and Finalization: Java Card does not require garbage collection. It also does not allow explicit deallocation of the objects as Java does. Finalization also is not required. The virtual machine will not call a finalize method automatically.

A. CARD PROGRAMMING CONCEPTS

Programming in Java Card is quite different from other applications. The following list of concepts is important in understanding a Java Card application [Ref. 4]. Some of the concepts come from a smart card standard promulgated by the International Standards Organization (ISO). In particular, they are part of the ISO 7816 standard.

1. Applications have unique identifiers as defined in ISO 7816-5. Each is called an *AID*, which stands for Application Identifier.

2. The card interfaces with the outside world via a data structure called the Application Protocol Data Unit, or *APDU*. It is described in ISO 7816-5.

3. The basic unit of execution on a Java Card is the *Applet*. It is the entry point for a service provided by the card.

4. *Applet Execution Context*. The JCRE keeps track of the currently selected Applet as well as the currently active Applet. The environment of the currently active Applet is referred to as the Applet Execution Context. When a virtual method is invoked on an object, the Applet execution context is changed to correspond to the Applet that owns that object. When the method returns, the previous context is restored. Invocations of the static methods have no effect on the Applet execution context. The Applet execution context and sharing status of an object together determine if access to that object is permitted.

5. *Card Acceptance Device*. A Java Card, like any smart card, gets inserted into a card terminal for its power supply and interface. This terminal is often called the Card Acceptance Device, or CAD.

6

6. *Atomic Operation*. This is an operation that can never be partially executed. Either it executes to completion or does not execute at all. The property is needed in order to guarantee invariant conditions in the presence of unexpected behavior, such as loss of power to the card after being removed from the CAD.

7. *Transaction*. A transaction is an atomic operation where the programmer defines the extent of the operation by indicating in the program code the beginning and end of the transaction.

8. *Java Card Runtime Environment*. This environment consists of the Java Card-Virtual Machine and the core classes in the Java Card API.

9. *Persistent Object*. A persistent object is one whose state persists between card insertions. Objects are persistent by default. Persistent object state is updated atomically using transactions. The term persistent does not mean there is an object-oriented database on the card or that objects are serialized/deserialized, just that the objects are not lost when the card loses power.

10. *Transient Object*. A transient object is one whose state is not saved across card insertions. Its state is reset to a default state at specified intervals. Updates to the state of transient objects are not atomic and are not affected by transactions.

B.   THE JAVA CARD FRAMEWORK

ISO 7816 or EMV (Europay, MasterCard, Visa) is the underlying standard on which Java Card 2.0 stands. ISO 7816 defines the standards for Integrated Circuit Cards (ICCs) with contacts, also known as smart cards. It covers the various aspects of the Smart Cards, and consists of six parts. The Java Card framework is designed to handle most of the low-level details specified in ISO 7816, parts 1-3. It also provides classes and methods that assist developers in being compatible with parts 4-6. The EMV standard, which is defined by the members of the international financial community, consists of a subset of ISO 7816 Part 1-6, with additional proprietary features which are required to meet the specific needs of the financial industry.

1.   Applets

Applets in Java Card correspond to the application in ISO 7816. They are the basic unit of selection, context, functionality, and security [Ref. 4]. They are

7

identified by an Application Identifier (AID) and selected by a CAD on demand. A CAD sends formatted commands as APDU buffers to Applets. Applets reply to each APDU command with optional data and indicate the results of the operation using a status word (SW) as defined in ISO 7816 Part-4. Applets define the behavior of an object when they are instantiated. They are a subclass of `javacard.system.Applet`, which defines the common behavior of all Applets. A package is a collection of Applets as in Java.

2.   Objects

Objects in Java Card have the same behavior as in Java. They are used to represent, store, and manipulate data by the Applets. An Applet which instantiates the object is able to use and modify the object. Objects created by an Applet can be shared with other Applets as long as the owner of the Applet permits sharing. The rules defining the lifetime of an object and its creation are as follows:

- The lifetime of an object depends on the existence of a pointer pointing to it. The pointer can be stored in a local or parameter variable or field of another object.

- Once an object is instantiated, all the fields of the object are set to their default values. For example, if the data type of a field is `int`, then its default value will be 0 as in Java.

An object is persistent if 1) the object is the subclass of the Applet class and registers itself, 2) the JCRE stores the reference to these objects to make them persistent, or 3) the object is a part of another persistent object. If the object is not registered or not referenced by any other persistent object, it can be discarded or garbage collected.

3.   Virtual Machine

In a PC or workstation, the Java Virtual Machine is a regular process. When the process dies, the resources of the object are deallocated. But the *execution lifetime* of the Java Card Virtual Machine is the lifetime of the card. The virtual machine does

not depend on any other power source and uses persistent memory technology (such as EEPROM). During the card initialization stage, the framework and the JCRE are installed and exist for the lifetime of the virtual machine[Ref. 4]. The execution lifetime of the JCRE and the framework span CAD sessions. Therefore, the execution lifetime of the objects created by the Applets of the framework spans CAD sessions as long as there is a reference to them.

4.   Transactions

Transactions are among the most important concepts in Java Card. The atomic transaction model of Java Card requires that the updates to the fields of an object take place correctly and consistently or else all fields are restored to their previous values. An Applet has the ability to mark the beginning and end of an atomic transaction. It also has the ability to undo all updates in the middle of a transaction if it encounters an internal problem or decides to cancel the transaction.

The current transaction model allows one transaction in progress at a time. If an attempt is made to enter a transaction within another transaction, then an exception is thrown. Java Card allows Applets to inspect whether a transaction is still in progress. If power should fail while a transaction is in progress, all fields updated since the start of the transaction are restored to their previous values. The restoration is done during re-initialization after the failure or reset by the JCRE. Transient objects are not restored upon re-initialization following an aborted transaction. In the case of an internal problem, an Applet can decide to abort the transaction by a method call. All the values updated since the beginning of the transaction are restored to their previous values, and the transaction flag is reset. If a `select`, `deselect` or `process` method is invoked while in a transaction, the JCRE will automatically abort the transaction.

Finally, the resources of a Java Card system are limited. The number of bytes of conditionally updated objects that can be accumulated during a transaction cannot exceed this limit. Java Card provides functions to query how much *commit*

capacity is available on the current platform. Exceeding the commit capacity causes an exception. The JCRE can choose to either mute the exception or make it visible to the interface.

5. Applet Isolation and Object Sharing

Applet isolation is provided by an Applet firewall which prevents one Applet from accessing objects owned by other Applets. The Applet that was active when the object was created owns the object. All the privileges to use and to modify the object belong to the owner. An Applet can have a reference to the object which is created by another Applet but it cannot invoke methods on the object or set the contents of its fields. On the other hand, the JCRE must be able to invoke methods on Applets, and Applets must be able to use objects owned by the JCRE. If an Applet does not have sharing privileges for an object, any attempt to invoke an instance method or access the objects's contents will throw a *Security Exception.*

The JCRE can modify any object on the card whether or not that object is shared. An Applet may permit unrestricted sharing of any of its objects. Once the object is shared, it is shared for its remaining lifetime. An Applet may also permit restricted sharing of any of its objects. Restricted sharing can be used when one wants to share an object with a certain Applet. An Applet can call the share method more than once to share the object with different Applets.

6. Applet Lifetime and Runtime Environment

An Applet lives as long as the card since it is loaded onto the card during card production. Applets are a subclass of the Applet class, as mentioned above. The JCRE interacts with the Applet via the Applet's public methods install, select, deselect, and process. The Applet must implement the install method. If the install method is not implemented, the Applet's objects cannot be created or initialized [Ref. 4].

After installation (calling the install method), an Applet is responsible for

its own state, which it determines by the way it responds to the invocation of its select, deselect, and process methods. Basically, Applets are completely responsible for their internal state. The mechanics of the Applet invocation are as follows. Any select APDU with the Applet's AID will cause this Applet's select method to be invoked. Any select APDU with another Applet's AID will cause this Applet's deselect method to be invoked. Any APDU other than select will cause this Applet's process method to be invoked. Once an Applet is selected, it stays selected until power is lost, the card is reset, or another Applet is selected. When the Applet is selected, its process method can maintain its own state (including states like blocked or expired), reference (read and write) its own objects, reference shared objects, share its objects with other Applets, enclose multiple updates in a transaction, create new objects (if the policy allows this), and invoke services provided by the Java Card Application Programming Interface, such as Personal Identification Number (PIN), cryptography, and file system services.

Power loss occurs when the card is withdrawn from the CAD. When power is re-supplied to the card , the JCRE ensures that all transient object fields are reset to their default state, the transaction in progress, if any, is aborted, and the Applet becomes deselected but the deselect method is not called.

7. The APDU

A Smart Card uses data units to communicate with the outside world. These units, called APDU buffers, contain either a command or a response message. A smart card system follows the master-slave model in which the smart card plays the passive role. In other words, a smart card always waits for a command APDU from a terminal. It then executes the action specified in the APDU and replies to the terminal with a response APDU. The command APDU has the format in Figure 1.

The header contains the coding of a command. It has four fields:

1. CLA Class byte. In many Smart Cards, this byte is used to identify an application.

11

Command APDU						
Mandotary Header				Conditional Body		
CLA	INS	P1	p2	Lc	Data Field	Le

Figure 1. Command APDU Format

2. INS Instruction byte. This byte indicates the instruction code.

3. P1-P2 Parameter bytes. These provide further qualification of the APDU command.

The conditional body has two fields other than the data field:

1. Lc denotes the number of bytes in the data field of the command APDU.

2. Le denotes the maximum number of bytes expected in the data field of the response APDU.

The response APDU has the format in Figure 2.

Response APDU		
Conditional Body	Mandatory Trailer	
Data Field	SW1	SW2

Figure 2. Response APDU Format

The status bytes SW1 and SW2 denote the processing status of the command APDU in a card. The Java Card APDU class provides methods to handle APDUs which conform to the ISO 7816 Part-4. It is also carefully designed to abstract the underlying transport protocol changes. We can summarize the responses of Java Card to the different APDU commands as follows. In the first case, there are no command data and no response data. In the second case, there are no command data, but there are response data. The details of these cases can be found in [Ref. 4].

III.   THE LANGUAGE AND TYPE SYSTEM

The secure flow type system covers a subset of Java Card 2.0. The major structures of the language are treated, however, some features have been omitted. For example, transactions have not been considered. Although committing a transaction is done through a method call, the implications of it, from a privacy standpoint, need to be investigated.

A.   THE LANGUAGE

In the following grammar, e denotes an *expression*, c a *command*, and b a function body. We use $x, y, \dots$ to denote identifiers and n to stand for an arbitrary integer.

$$
\begin{aligned}
p \;::=&\; e \mid c \\
e \;::=&\; n \mid x \mid e_1 + e_2 \mid x(e, e') \mid x[e] \mid \textbf{new int}[e] \\
b \;::=&\; e;\, e' \mid e_1 := e_2 \mid \textbf{int } x := e;\, e' \mid \\
 &\; \textbf{int[ ] } x := e;\, e' \mid \textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2 \\
c \;::=&\; e_1 := e_2 \mid \textbf{if } e \textbf{ then } c \textbf{ else } c' \mid \\
 &\; \textbf{C } x := \textbf{new C}(e, e');\, c \mid \textbf{int } x := e;\, c \mid c;\, c' \mid \\
 &\; \textbf{int[ ] } x := e;\, c \mid \textbf{int } z(\textbf{int } x,\ \textbf{int[] } y)\{b\}\, c \mid \\
 &\; \textbf{void } z(\textbf{int } x,\ \textbf{int[] } y)\{c\}\, c' \mid x(e, e')
\end{aligned}
$$

Notice that the body b of a function

$$
\textbf{int } z(\textbf{int } x,\ \textbf{int[] } y)\{b\}
$$

is not a command. This means that no procedure call is allowed in a function body. A function body is an extension of the expressions that allows some commands to be used and typed as expressions. Consequently, special consideration is needed in typing function method declarations, as we shall see, in order to preserve an important security property of well-typed programs called *Confinement*.

13

An array declaration

$$\textbf{int}[]\ x\ :=\ e;\ c$$

declares an array reference variable x and initializes it to the reference value obtained by evaluating expression e. The expression e has the form

$$\textbf{new int}[e']$$

for some expression e'. Array references are first-class values, whereas variables are not. The declaration

$$\texttt{C}\ x\ :=\ \textbf{new}\ \texttt{C}(e, e');\ c$$

declares a reference x to a C object. The language differs from Java Card 2.0 in that the right side of the declaration is not an expression. We couple object creation with variable declaration. Objects cannot be created by executing an expression. They can be created only in the context of an object variable declaration. Further, we allow at most one object to be created for a given class. This means that methods are not polymorphic like procedures in [Ref. 1].

B. TYPING RULES

The types of the secure flow type system are given below:

$$\tau\ ::=\ L\ |\ H$$
$$\pi\ ::=\ \tau\ |\ \tau\ proc(\tau, \tau\ arr)\ |\ \tau\ fun(\tau, \tau\ arr)\ |\ \tau\ arr\ fun(\tau, \tau\ arr)$$
$$\rho\ ::=\ \pi\ |\ \tau\ var\ |\ \tau\ arr\ |\ \tau\ arr\ var\ |\ \tau\ cmd$$

Every integer has a type τ which for our purposes is a security level high (H) or low (L). A constant array reference has type $\tau\ arr$ and is a single-dimensional array. Functions return either an integer or an array reference and have types $\tau\ fun$ and $\tau\ arr\ fun$ respectively. Variables have type $\tau\ var$ and a variable that stores an array reference has type $\tau\ arr\ var$. (Notice that we distinguish an array reference from an array reference variable.)

14

$$(\text{IDENT}) \qquad \gamma \vdash x : \tau \quad \gamma(x) = \tau$$

$$(\text{VAR}) \qquad \gamma \vdash x : \tau \; var \qquad \gamma(x) = \tau \; var$$

$$(\text{ARRREF}) \qquad \gamma \vdash x : \tau \; arr \qquad \gamma(x) = \tau \; arr$$

$$(\text{ARRREFVAR}) \qquad \gamma \vdash x : \tau \; arr \; var \qquad \gamma(x) = \tau \; arr \; var$$

$$(\text{INT}) \qquad \gamma \vdash n : \tau$$

$$(\text{R-VAL}) \qquad \frac{\gamma \vdash e : \tau \; var}{\gamma \vdash e : \tau}$$

$$\frac{\gamma \vdash e : \tau \; arr \; var}{\gamma \vdash e : \tau \; arr}$$

$$(\text{SUM}) \qquad \frac{\gamma \vdash e : \tau, \quad \gamma \vdash e' : \tau}{\gamma \vdash e + e' : \tau}$$

$$(\text{NEW}) \qquad \frac{\gamma \vdash e : \tau}{\gamma \vdash \textbf{new int}[e] : \tau \; arr}$$

$$(\text{RETURN}) \qquad \frac{\gamma \vdash e : \tau}{\gamma \vdash \textbf{return } e : \tau}$$

$$(\text{ARR-INDEX}) \qquad \frac{\gamma \vdash x : \tau \; arr, \quad \gamma \vdash e' : \tau}{\gamma \vdash x[e] : \tau \; var}$$

$$(\text{COMPOSE}) \qquad \frac{\gamma \vdash e : \tau, \quad \gamma \vdash e' : \tau}{\gamma \vdash e; e' : \tau}$$

$$(\text{ASSIGN}) \qquad \frac{\gamma \vdash e : \tau \; var, \quad \gamma \vdash e' : \tau}{\gamma \vdash e := e' : \tau}$$

$$(\text{LETVAR}) \qquad \frac{\gamma \vdash e : \tau, \quad \gamma[x : \tau \; var] \vdash e' : \tau'}{\gamma \vdash \textbf{int } x := e \; ; \; e' : \tau'}$$

$$(\text{LETARRVAR1}) \qquad \frac{\gamma \vdash e : \tau \; arr, \quad \gamma[x : \tau \; arr \; var] \vdash e' : \tau'}{\gamma \vdash \textbf{int}[] \; x := e \; ; \; e' : \tau'}$$

$$(\text{FUNCALL}) \qquad \frac{\begin{array}{c} \gamma(x) = \tau \; fun(\tau, \tau \; arr) \\ \gamma \vdash e : \tau \\ \gamma \vdash e' : \tau \; arr \end{array}}{\gamma \vdash x(e, e') : \tau}$$

$$(\text{IF}) \qquad \frac{\gamma \vdash e : \tau, \quad \gamma \vdash e' : \tau, \quad \gamma \vdash e'' : \tau,}{\gamma \vdash \textbf{if } e \textbf{ then } e' \textbf{ else } e'' : \tau}$$

Figure 3. Typing Rules for Expressions

(RETURN) $\gamma \vdash \textbf{return} : \tau \ cmd$

(COMPOSE) $$\frac{\gamma \vdash c : \tau \ cmd, \ \ \gamma \vdash c' : \tau \ cmd}{\gamma \vdash c ; c' : \tau \ cmd}$$

(ASSIGN) $$\frac{\gamma \vdash e : \tau \ arr \ var, \ \ \gamma \vdash e' : tau \ arr}{\gamma \vdash e := e' : \tau \ cmd}$$

(LETVAR) $$\frac{\gamma \vdash e : \tau, \ \ \gamma[x : \tau \ var] \vdash c : \tau' \ cmd}{\gamma \vdash \textbf{int} \ x := e \ ; \ c : \tau' \ cmd}$$

(LETARRVAR2) $$\frac{\gamma \vdash e : \tau \ arr, \ \ \gamma[x : \tau \ arr \ var] \vdash c : \tau' \ cmd}{\gamma \vdash \textbf{int}[] \ x := e \ ; \ c : \tau' \ cmd}$$

(LETFUN) $$\frac{\begin{array}{l} \gamma[x : \tau', y : \tau' \ arr] \vdash b : \tau' \\ b \ is \ pure \ with \ respect \ to \ \gamma[y : \tau' \ arr] \\ \gamma[z : \tau' \ fun(\tau', \tau' \ arr)] \vdash c : \tau \ cmd \end{array}}{\gamma \vdash \textbf{int} \ z(\textbf{int} \ x, \textbf{int}[] \ y) \ \{b\} \ c : \tau \ cmd}$$

(LETPROC) $$\frac{\begin{array}{l} \gamma[x : \tau', y : \tau'' \ arr] \vdash c : \tau''' \ cmd \\ \gamma[z : \tau''' \ proc(\tau', \tau'' \ arr)] \vdash c' : \tau \ cmd \end{array}}{\gamma \vdash \textbf{void} \ z(\textbf{int} \ x, \textbf{int}[] \ y) \ \{c\} \ c' : \tau \ cmd}$$

(PROCCALL) $$\frac{\begin{array}{l} \gamma(x) \ = \ \tau \ proc(\tau', \tau'' \ arr) \\ \gamma \vdash e : \tau' \\ \gamma \vdash e' : \tau'' \ arr \end{array}}{\gamma \vdash x(e, e') : \tau \ cmd}$$

(IF) $$\frac{\gamma \vdash e : \tau, \ \ \gamma \vdash c : \tau \ cmd, \ \ \gamma \vdash c' : \tau \ cmd,}{\gamma \vdash \textbf{if} \ e \ \textbf{then} \ c \ \textbf{else} \ c' : \tau \ cmd}$$

(WHILE) $$\frac{\gamma \vdash e : \tau, \ \ \gamma \vdash c : \tau \ cmd}{\gamma \vdash \textbf{while} \ e \ \textbf{do} \ c : \tau \ cmd}$$

(LETOBJECT) $$\frac{\begin{array}{l} \gamma(\texttt{C}) \ = \ \tau \ proc(\tau', \tau'' \ arr) \\ \gamma \vdash e : \tau' \\ \gamma \vdash e' : \tau'' \ arr \\ \gamma \vdash c : \tau \ cmd \end{array}}{\gamma \vdash \texttt{C} \ x := \textbf{new} \ \texttt{C} \ (e, e') \ ; \ c : \tau \ cmd}$$

Figure 4. Typing Rules for Commands

16

The typing rules are given in Figures 3 and 4. Notice how array indexing is typed in rule ARR-INDEX. The security level of the index and the array reference must be the same. That means that arrays can store only information whose security level is equal to the array reference.

Array declarations have two different rules, one (LETARRVAR1) for expression contexts and the other (LETARRVAR2) for command contexts. These rules introduce array references as first-class values. Function declaration has an unusual typing rule. We require body b to be pure with respect to γ, which means that for all $y \in dom(\gamma)$, y is not updated in b if y is free in b and either $\gamma(y) = L\ var$ or $\gamma(y) = L\ arr$. In other words, all assignments in a function body involve assignments to local variables only unless their security levels are high.

C. CONFINEMENT

Java Card 2.0 inherits the assignment expression, pre/post increment/decrement and conditional expressions from Java. These structures cause side effects in an expression, which can violate an important property called *Confinement* if they are not typed appropriately. Basically, *Confinement* says that the execution of any high command does not result in updating any low variables. For example, suppose that x is a high boolean variable and y is a low boolean variable, and consider

$$\text{if } x \text{ then } x := (y := 1)$$

If the expression $(y := 1)$ can be typed as a high expression and the command $x := (y := 1)$ can be typed as a high command, then the whole conditional can be typed as a high command. But y is a low variable and the execution of the conditional causes a low variable to be updated, which is a violation of *Confinement*. To preserve Confinement, the type system demands that function bodies be pure. Other Java Card expressions like pre/post increment/decrement could easily be added to the language without affecting the type system as long as they are limited to function bodies where the typing rule for function declarations comes into play.

17

IV.   THE SECURE FLOW TYPE INFERENCE ALGORITHM

The secure flow type inference algorithm is based on the algorithm of Volpano and Smith given in [Ref. 1]. It is given for the language and the type system described in Chapter III.

A.   THE TYPE INFERENCE ALGORITHM

The type inference algorithm W has the following inputs and outputs:

- Inputs

  1. $\hat{\gamma}$: a type environment – maintains types of free identifiers and variables.
  2. p : a program phrase – the phrase to be typed.
  3. V : a set of stale type variables – empty initially.

- Outputs

  1. C : a constraint set – consists of inequalities of the form $\tau \leq \tau'$. A constraint $\tau = \tau'$ is understood to mean $\tau \leq \tau'$ and $\tau' \leq \tau$.
  2. $\hat{\pi}$: a π type with type variables – the result type of the phrase p.
  3. V' : a set of stale type variables – $(V' - V)$ is the set of type variables generated during the typing of phrase p.

The algorithm is given by cases in Figures 5, 6 and 7. Each case corresponds to a different phrase of the language.

Notice that the rule for an expression $x[e]$ requires that the security level of the index and the array reference be the same. Naturally, we add a constraint that forces them to be equal, specifically $\hat{\tau} = \hat{\tau}_1$. Also notice the coercions in several places in the algorithm. They have the form $\hat{\tau} \leq \alpha$ or $\alpha \leq \hat{\tau}$ for some fresh type variable α. The first is an upward coercion reflecting the idea that a low producer (expression) may be regarded as a high producer. The second is a downward coercion and effectively reflects the idea that a high consumer (command) can be a low consumer. Algorithm

$W(\widehat{\gamma}, p, V) = $ case p of

x : case $\widehat{\gamma}(x)$ of

$\qquad \widehat{\tau} : (\{\widehat{\tau} \leq \alpha\}, \alpha, V \cup \{\alpha\}) \quad \alpha \notin V$

$\qquad \widehat{\tau}\ var : (\{\widehat{\tau} \leq \alpha\}, \alpha, V \cup \{\alpha\}) \quad \alpha \notin V$

$\qquad \widehat{\tau}\ arr : (\{\ \}, \widehat{\tau}\ arr, V \cup \{\ \})$

$\qquad \widehat{\tau}\ arr\ var : (\{\ \}, \widehat{\tau}\ arr, V \cup \{\ \})$

$\qquad$ default : fail

$n : (\{\ \}, \alpha, V \cup \{\alpha\}) \quad \alpha \notin V$

$e_1 + e_2 :$

$\qquad$ let $(C_1, \widehat{\tau}_1, V') = W(\lambda, \widehat{\gamma}, e_1, V)$

$\qquad$ let $(C_2, \widehat{\tau}_2, V'') = W(\lambda, \widehat{\gamma}, e_2, V')$

$\qquad$ in $(C_1 \cup C_2 \cup \{\widehat{\tau}_1 = \widehat{\tau}_2\}, \widehat{\tau}_1, V'')$

$x[e] :$

$\qquad$ let $(C', \widehat{\tau}_1, V') = W(\widehat{\gamma}, e, V)$

$\qquad$ case $\widehat{\gamma}(x)$ of

$\qquad\qquad \widehat{\tau}\ arr\ var : (C' \cup \{\widehat{\tau} = \widehat{\tau}_1, \widehat{\tau} \leq \alpha\}, \alpha, V \cup \{\alpha\}) \quad \alpha \notin V$

$\qquad\qquad \widehat{\tau}\ arr : (C' \cup \{\widehat{\tau} = \widehat{\tau}_1, \widehat{\tau} \leq \alpha\}, \alpha, V \cup \{\alpha\}) \quad \alpha \notin V$

$\qquad\qquad$ default : fail

$e_1 := e_2 :$

$\qquad$ case e_1 of

$\qquad\qquad x$: if $\widehat{\gamma}(x) = \widehat{\tau}\ var$ then

$\qquad\qquad\qquad$ let $(C, \widehat{\tau}', V') = W(\widehat{\gamma}, e_2, V)$ in

$\qquad\qquad\qquad\qquad (C \cup \{\widehat{\tau} = \widehat{\tau}', \alpha \leq \widehat{\tau}'\}, \alpha\ cmd, V' \cup \{\alpha\}) \quad \alpha \notin V'$

$\qquad\qquad\qquad$ elsif $\widehat{\gamma}(x) = \widehat{\tau}\ arr\ var$ then

$\qquad\qquad\qquad\qquad$ let $(C, \widehat{\tau}'\ arr, V') = W(\widehat{\gamma}, e_2, V)$ in

$\qquad\qquad\qquad\qquad\qquad (C \cup \{\widehat{\tau} = \widehat{\tau}', \alpha \leq \widehat{\tau}'\}, \alpha\ cmd, V' \cup \{\alpha\}) \quad \alpha \notin V'$

$\qquad\qquad\qquad$ else fail

$\qquad\qquad x[e]$: if $\widehat{\gamma}(x) = \widehat{\tau}\ arr\ var$ or $\widehat{\gamma}(x) = \widehat{\tau}\ arr$ then

$\qquad\qquad\qquad\qquad$ let $(C, \widehat{\tau}', V') = W(\widehat{\gamma}, e_2, V)$ in

$\qquad\qquad\qquad\qquad\qquad (C \cup \{\widehat{\tau} = \widehat{\tau}', \alpha \leq \widehat{\tau}'\}, \alpha\ cmd, V' \cup \{\alpha\}) \quad \alpha \notin V'$

$\qquad\qquad\qquad$ else fail

$\qquad\qquad$ default : fail

new $native[e] :$

$\qquad$ let $native ::= $ short $|$ boolean $|$ byte

$\qquad\qquad$ $let(C, \widehat{\tau}, V') = W(\widehat{\gamma}, e, V)$ in $(C, \widehat{\tau}\ arr, V')$

Figure 5. Algorithm W

if e **then** c_1 **else** c_2 :

       let $(C, \hat{\tau}, V') = W(\hat{\gamma}, e, V)$

       let $(C_1, \hat{\tau}_1 \; cmd, V'') = W(\hat{\gamma}, c_1, V')$

       let $(C_2, \hat{\tau}_2 \; cmd, V''') = W(\hat{\gamma}, c_2, V'')$

       in $(C \cup C_1 \cup C_2 \cup \{\hat{\tau} = \hat{\tau}_1 = \hat{\tau}_2, \alpha \leq \hat{\tau}\}, \alpha \; cmd, V''' \cup \{\alpha\})$ $\alpha \notin V'''$

while e **do** c :

       let $(C, \hat{\tau}, V') = W(\hat{\gamma}, e, V)$

       let $(C', \hat{\tau}' \; cmd, V'') = W(\hat{\gamma}, c, V')$

       in $(C \cup C' \cup \{\hat{\tau} = \hat{\tau}', \alpha \leq \hat{\tau}\}, \alpha \; cmd, V'' \cup \{\alpha\})$ $\alpha \notin V''$

$c_1; \; c_2$:

       let $(C_1, \hat{\tau}_1 \; cmd, V') = W(\hat{\gamma}, c_1, V)$

       let $(C_2, \hat{\tau}_2 \; cmd, V'') = W(\hat{\gamma}, c_2, V')$

       in $(C_1 \cup C_2 \cup \{\hat{\tau}_1 = \hat{\tau}_2\}, \hat{\tau}_1 \; cmd, V'')$

int $x := e$; c :

       let $(C, \hat{\tau}, V') = W(\hat{\gamma}, e, V)$

       let $(C', \hat{\tau}' \; cmd, V'') = W(\hat{\gamma}[x : \hat{\tau} \; var], c, V')$

       in $(C \cup C', \hat{\tau}' \; cmd, V'')$

int $x := e$; e' :

       let $(C, \hat{\tau}, V') = W(\lambda, \hat{\gamma}, e, V)$

       let $(C', \hat{\tau}', V'') = W(\lambda, \hat{\gamma}[x : \hat{\tau} \; var], e', V')$

       in $(C \cup C', \hat{\tau}', V'')$

int[] $x := e$; c :

       let $(C, \hat{\tau} \; arr, V') = W(\lambda, \hat{\gamma}, e, V)$

       let $(C', \hat{\tau}' \; cmd, V'') = W(\lambda, \hat{\gamma}[x : \hat{\tau} \; arr \; var], c, V')$

       in $(C \cup C', \hat{\tau}' \; cmd, V'')$

int[] $x := e$; e' :

       let $(C, \hat{\tau} \; arr, V') = W(\lambda, \hat{\gamma}, e, V)$

       let $(C', \hat{\tau}', V'') = W(\lambda, \hat{\gamma}[x : \hat{\tau} \; arr \; var], e', V')$

       in $(C \cup C', \hat{\tau}', V'')$

int z(**int** x, **int**[] y) $\{b\}$ c :

       let $(C, \hat{\tau}, V') = W(\lambda, \hat{\gamma}[x : \hat{\tau}, y : \hat{\tau} \; arr], b, V)$

       let $(C', \hat{\tau}' \; cmd, V'') = W(\lambda, \hat{\gamma}[z : \hat{\tau} \; fun(\hat{\tau}, \hat{\tau} \; arr)], c, V')$

       in $(C \cup C', \hat{\tau}' \; cmd, V'')$

Figure 6. Algorithm W, continued

void $z(\textbf{int } x, \textbf{int}[] \ y) \ \{c\} \ c'$:

$\quad$ let $(C, \hat{\tau}'' \ cmd, V') = W(\lambda, \hat{\gamma}[x : \hat{\tau}, y : \hat{\tau}' \ arr], c, V)$
$\quad$ let $(C', \hat{\tau}''' \ cmd, V'') = W(\lambda, \hat{\gamma}[z : \hat{\tau}'' \ proc(\hat{\tau}, \hat{\tau}' \ arr)], c', V')$
$\quad$ in $(C \cup C', \hat{\tau}''' \ cmd, V'')$

C $x := \textbf{new}$ C (e, e') ; c :

$\quad$ if $\hat{\gamma}(\texttt{C}) = \hat{\tau}'' \ proc(\hat{\tau}, \hat{\tau}' \ arr)$
$\quad\quad$ let $(C, \hat{\tau}_1, V') = W(\lambda, \hat{\gamma}, e, V)$
$\quad\quad$ let $(C', \hat{\tau}_2 \ arr, V'') = W(\lambda, \hat{\gamma}, e', V')$
$\quad\quad$ let $(C'', \hat{\tau}'' \ cmd, V''') = W(\lambda, \hat{\gamma}', c, V'')$
$\quad\quad$ in $(C \cup C' \cup C'' \cup \{\hat{\tau} = \hat{\tau}_1, \hat{\tau}' = \hat{\tau}_2\}, \hat{\tau}'' \ cmd, V''')$
$\quad$ else fail

$x(e, e')$:

$\quad$ if $\hat{\gamma}(x) = \hat{\tau}'' \ proc(\hat{\tau}, \hat{\tau}' \ arr)$
$\quad\quad$ et $(C, \hat{\tau}_1, V') = W(\lambda, \hat{\gamma}, e, V)$
$\quad\quad$ let $(C', \hat{\tau}_2 \ arr, V'') = W(\lambda, \hat{\gamma}, e', V')$
$\quad\quad$ in $(C \cup C' \cup C'' \cup \{\hat{\tau} = \hat{\tau}_1, \hat{\tau}' = \hat{\tau}_2\}, \hat{\tau}'' \ cmd, V''')$
$\quad$ elsif $\hat{\gamma}(x) = \hat{\tau} \ fun(\hat{\tau}, \hat{\tau} \ arr)$
$\quad\quad$ let $(C, \hat{\tau}_1, V') = W(\lambda, \hat{\gamma}, e, V)$
$\quad\quad$ let $(C', \hat{\tau}_2 \ arr, V'') = W(\lambda, \hat{\gamma}, e', V')$
$\quad\quad$ in $(C \cup C' \cup C'' \cup \{\hat{\tau} = \hat{\tau}_1, \hat{\tau}_1 = \hat{\tau}_2\}, \hat{\tau}, V''')$
$\quad$ else fail

Figure 7. Algorithm W, continued

W on a function method call, where the function returns an array reference, is defined like it is for functions returning integers. The return type becomes $\hat{\tau} \ arr$. Therefore, this case is omitted from the specification of W in Figure 7.

B. AN IMPLEMENTATION IN JAVACC

The type inference algorithm has been implemented using a compiler-compiler called *JavaCC*. JavaCC generates a top-down parser for a given syntactic/semantic specification as input. For us, the semantic component is the secure flow type inference algorithm. The generated parser implements the algorithm above. Starting with a JavaCC specification for Java 1.0.2, a specification was built for the core language of Chapter III. We needed to make many changes in the specification since the language

in Chapter III is a significant subset of Java 1.0.2. The result was a secure-flow analyzer for most of Java Card 2.0. In Chapter V, we analyze a sample Card Applet using the analyzer.

V. APPLICATION OF THE SECURE FLOW ANALYZER

Now we shall take a look at a sample Java Card Applet using the Java Card 2.0 API and our secure flow analyzer in order to determine whether or not the Applet leaks any information about a private crypto key. We have the following scenario. A smart card comes with a preloaded decryption library and a crypto key. The sample Applet's task is to receive encrypted data from the CAD and then submit it to a decryption method in a library. The decryption method will use the key to decrypt the data. To differentiate the tasks, we will refer to the sample Applet as the `Client` Applet and the decryption library as the `Decrypt` class.

A. THE CLIENT APPLET

The Client Applet is given in Figure 8. It uses a variety of different methods from the Java Card 2.0 API:

- `install()`

  1. Create the Applet's instance.
  2. Register the Applet (by calling the constructor).

- `select()`

  1. Select the Applet.
  2. Return *true* to guarantee selection in this case.

- `process()`

  1. Check that the correct commands are received from the CAD.
  2. Send the encrypted data to the library.

```
import javacard.framework.*;

public class Client extends Applet {

  public static final byte CLIENT_CLA = (byte) 0x80;
  public static final byte CLIENT_INS = (byte) 0x20;

  private byte[] buffer;
  private decrypt;

  public Client() {

    decrypt = Decrypt.getDecrypt();
    register();
  }

  public boolean select() { return true; }

  public static void install(APDU apdu) throws ISOException {

    new Client();
  }

  public void process(APDU apdu) throws ISOException {

    buffer = apdu.getBuffer();

    if ( CLIENT_CLA != buffer[ISO.OFFSET_CLA] )
      ISOException.throwIt(ISO.SW_CLA_NOT_SUPPORTED);
    if ( CLIENT_INS != buffer[ISO.OFFSET_INS] )
      ISOException.throwIt(ISO.SW_INS_NOT_SUPPORTED);

    apdu.setIncomingAndReceive();

    decrypt.decrypt(buffer);
  }
}
```

Figure 8. Client.java

The Decrypt class is given in Figure 9. Its methods are described below.

- getDecrypt()

  1. Return the single instance of the Decrypt class.

- decrypt()

  1. Check whether key is set or not.
  2. Check whether the input data buffer size is bigger than the expected size.
  3. Decrypt the data.

```java
import javacard.framework.*;

public class Decrypt {

  private static final short KEY_NOT_SET = 0xFF01;
  private static byte[] buffer = new byte[50];

  private Decrypt() {}
  private static final Decrypt decrypt = new Decrypt();
  public static Decrypt getDecrypt() { return decrypt; }

  public void decrypt(byte[] data) {

    if (key == 0) {
      ISOException.throwIt(KEY_NOT_SET);
    } else {
      if (data[ISO.OFFSET_LC] > buffer.length)
        ISOException.throwIt(ISO.SW_WRONG_LENGTH);
      // Start decryption - End decryption
    }
  }
}
```

Figure 9. Decrypt.java

B. THE TYPE ENVIRONMENT FOR THE API

In order to use the analyzer, we need to build an initial type environment that gives the types of methods and identifiers occuring free in the Applet:

1. `apdu.setIncomingAndReceive()`: *L proc()*– this is the primary *receive* method[Ref. 5]. It returns the number of bytes that can fit in the APDU buffer and also transfers the bytes from the CAD to the APDU buffer. Since we assume all data from the CAD is *low*, this method is typed *low*. Also, this method cannot be typed as a function, as in Java Card 2.0, since it has side effects.

2. `apdu.setOutgoingAndSend()`: *L proc(L,L)*– this is the send method. It sends `len` bytes in the APDU buffer, starting from `off`, to the CAD. Since the CAD is *low*, the method is *low*. The parameters of the method are also *low*, since they can be observed in the response APDU.

3. `apdu.getBuffer()`: *L arr func()*– this method returns a byte array containing the APDU buffer. We always regard this buffer as *low* since it can be seen at the CAD.

4. `ISOException.throwIt(sw)`: *L proc(L)*– throws the JCRE instance of the `ISOException` class with the specified status word `sw`. The parameter is typed *low* since the status word can appear as `sw1` and `sw2` of a response APDU.

5. `key`: *H*– a private crypto key.

6. `register()`: *H proc()*– this method is final and inherited from `Applet` class. It registers an Applet with the JCRE and appears not to involve any updates of *low* data structures. Although this needs to be confirmed, we shall go ahead and type its call as a *high* command.

The typings of the methods in the *Decrypt* class must be included in the initial type environment since they are free in the Applet. Therefore, we first analyze the `Decrypt` class and then merge the typings of this class with our initial type environment in order to analyze the `Client` Applet. But the analyzer fails on the `Decrypt` class and for good reason. In Java Card 2.0, user exceptions cause the control of the program to be transferred to the JCRE. Such an exception arises if the `key` is not set (key == 0) in our example. The exception KEY_NOT_SET reaches the JCRE where it can be observed at the interface (card reader). The presence of

28

the exception reveals whether key is zero, a violation of secure information flow. The secure flow analyzer requires the guard of the conditional, where key is checked, to be *low*. But key is *high*, so the analyzer produces an unsatisfiable constraint set with the inconsistent constraints

$$(\texttt{<= HIGH 8)} \quad \texttt{(= 8 LOW)}$$

The results of the analyzer are given in Figure 10. The analyzer generates a constraint set and a typing for each field and method of the Decrypt class in Figure 9.

```
Identifier        : Decrypt.KEY_NOT_SET
Type              : 2
Constraint Set    :


Identifier        : Decrypt.buffer
Type              : 4 arr var
Constraint Set    :


Identifier        : Decrypt
Type              : 5 proc()
Constraint Set    :


Identifier        : Decrypt.decrypt
Type              : 5
Constraint Set    :


Identifier        : getDecrypt
Type              : 5 func()
Constraint Set    :


Identifier        : decrypt
Type              : 16 proc(7 arr)
Constraint Set    : (<= HIGH 8)(= 8 9)(= 8 LOW)(<= 2 10)(= LOW 10)
                    (= LOW 15)(= 7 11)(<= 11 12)(<= LOW 11)
                    (<= LOW 13)(= 7 13)(= 7 LOW)(<= LOW 14)
                    (= LOW 14)(<= 15 7)(<= 16 8)
```

Figure 10. Results of the Analysis

VI. CONCLUSION

Protecting private information stored on smart cards is important to smart-card manufacturers who would like to provide some sort of guarantees about privacy to card issuers. The approach taken in this thesis is based on work by Volpano and Smith, who introduced a type system for privacy in a procedural programming language [Ref. 1]. Extensions of their type system are proposed to handle object creation, function methods, and first-class array references in an object-oriented programming language called Java Card. Java Card is gaining acceptance among major smart-card manufacturers as an open platform for smart card applications. The type system has been implemented as a Java Card program analyzer and demonstrated on a sample smart-card application.

The type system approach to privacy depends heavily upon the correctness of the inital type environment. The analyzer uses the initial type environment to get the security levels of variables that are free in the source code. Setting up an initial environment can be quite hard since you need to know the behavior of methods in the programming library, especially their interactions with the underlying operating system and hardware. The Java Card 2.0 specification strives to insulate card applications from low-level implementation details via a relatively abstract programming interface called the Application Programming Interface (API). This leaves some freedom to implement the API differently among the manufacturers. But then one cannot say for sure that these different implementations will preserve the semantic assumptions used in determining the initial type environment. One really does need details of an underlying implementation in order to get the environment correct. Often, though, such details are unavailable.

The type system approach also seems to be at odds with the ISO-7816 standard. It specifies the APDU as the interface between a smart card and a card reader, or acceptance device. Since the APDU buffer can be observed by anybody who has

31

access to a card reader, it should be regarded as low. On the other hand, sometimes it is used to hold private information entered via the reader by the card holder. So is the APDU buffer high or low? In essence, it is simultaneously a high and low variable! A more secure architecture would split the buffer into a private input channel and a public output channel which are much easier to treat in the type system. Smart cards with private LCD displays and private keypads seem more appropriate. This view is consistent with work done in exploring how smart cards can be used securely in a hostile environment [Ref. 6].

A. FUTURE WORK

The type system described in this thesis imposes some rather strong restrictions on Java Card Applets. First, function method bodies must be pure (no updates of free variables are allowed unless the variables are high). This restriction is hard to remove without giving up Confinement.

Second, object creation is restricted in that only one instance of a class can be created by an Applet. Further, the type system does not assign types to objects and object references are not typed as first-class references like arrays. The difficulty here is assigning a security level to an object when its fields may need different security levels. Taking the object's type to be the least upper bound of these levels might be too coarse. Allowing more than one instance of a class also means methods of the class must be typed polymorphically, which is more difficult to implement [Ref. 1]. There is also a problem with introducing object creation as an expression in the language since it involves executing a constructor that may have side effects. But allowing more than one instance of a class may not be worth the effort. If you look at the Java Card 2.0 specification, you will notice that the JCRE owns one instance of each exception and programmers are advised to use these instances in order to save on resources. Owning only one instance of a class is actually common in Applets. So limiting Applets to one instance of a class doesn't look like it will be a problem in

32

practice.

Lastly, dynamic method dispatching is prohibited because the type system needs to know the secure flow typing of a method call at compile time. Dynamic dispatching has a major impact on the type system. More experience should tell whether dynamic dispatching is really needed.

LIST OF REFERENCES

[1] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. *Proc. Theory and Practice of Software Development*, 1214:607–621, 1997.

[2] James D. Harvey. A static secure flow analyzer for java programs. Master's thesis, Naval Postgraduate School, 1998.

[3] Inc. Sun Microsystems. Java card 2.0 language subset and virtual machine specification, October 13 1997.

[4] Inc. Sun Microsystems. Java card 2.0 programming concepts, October 15 1997.

[5] Inc. Sun Microsystems. Java card 2.0 application programming interfaces, October 13 1997.

[6] J.D.Tygar Howard Gobioff, Sean Smith and Bennet Yee. Smart cards in hostile environments. *Proc. 2nd Usenix Workshop on Electronic Commerce*, pages 23–28, 1996.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center.....................................2
   8725 John J. Kingman Road., Ste 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library.......................................2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Deniz Kuvvetleri Komutanligi.......................................2
   Personel Daire Baskanligi
   Bakanliklar
   Ankara, TURKEY

4. Deniz Harp Okulu Komutanligi.......................................1
   Kutuphane
   Tuzla, Istanbul, TURKEY 81704

5. Chairman, Code CS.......................................1
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

6. Dennis Volpano, Code CS/VO.......................................2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5100

7. Craig Rasmussen, Code MA/RA.......................................1
   Mathematics Department
   Naval Postgraduate School
   Monterey, CA 93943-5100

8. Ismail Okan Akdemir.......................................2
   Cumhuriyet Mahallesi
   1. Yesil Sokak No:55/3
   45400 Turgutlu/MANISA/TURKEY